INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimizing a CFD Fortran code for GRID Computing

Stephen Wornom

N° 0303 – version 2

version initiale March 2005 – version révisée May 2005

Thème NUM

ISSN 0249-0803

INRIA
SOPHIA ANTIPOLIS

Optimizing a CFD Fortran code for GRID Computing

Stephen Wornom*

Thème NUM — Systèmes numériques
Projet SMASH

Rapport technique n° 0303 – version 2†— version initiale March 2005 — version révisée May 2005 29 pages

Abstract: Computations on clusters and computational GRIDS encounter similar situations where the processors used have different speeds and local RAM. In order to have efficient computations with processors of different speeds and local RAM, load balancing is necessary. That is, faster processors are given more work or larger domains to compute than the slower processors so that all processors finish their work at the same time thus avoiding faster processors waiting for the slower processors to finish. In addition, the programming language must permit dynamic memory allocation so that the executable size is proportional to the size of the partitions. The present version of the *AERO* code uses the F77 programming language which does not have dynamic memory allocation thus the size of the executable is the same for all processors and leads to situations where the RAM for some processors is too small to run the executable. In this report, we extend the parallel F77 *AERO* code to F90 which has dynamic memory allocation. The F90 version of the *AERO* code is mesh independent and because memory is allocated at runtime and memory is only allocated for the code options actually used, the size of the F90 executable is much smaller than the F77 version; as a consequence many tests cases that cannot be run on clusters and computational GRIDS with the F77 version can be easily run with the F90 version. Numerical results for a mesh containing 252K vertices using 8-*nina* and 8-*pf* processors running on the MecaGRID using GLOBUS using GLOBUS and heterogeneous partitions

* INRIA, 2004 Route des Lucioles, BP. 93, 06902 Sophia-Antipolis, France
† Title changed, added results for heterogeneous partitioning, added detailed description of the parallelization implementation

resulted in a speedup of 1.45 relative to the same run using the homogeneous partitioning which compares well with the theoretical speedup of 1.5. This validates the efficiency of the F90 version of the *AERO* code.

Key-words: Computational fluid dynamics, GRID Computing, F90

Optimisation d'un code fluide en Fortran sur Grille de Calcul

Résumé : Les calculs sur des clusters et des Grilles de calculs rencontrent des situations où les processeurs utilisés ont différentes vitesses et RAM locales. Afin d'avoir des calculs efficaces avec des processeurs de différentes vitesses et de RAM locale, l'équilibrage de charge est nécessaire. Des unités de traitement plus rapides reçoivent plus de travail ou de plus grands domaines du calcul que les processeurs plus lents de sorte que tous les processeurs finissent leur travail en même temps évitant de ce fait que les unités de traitement plus rapides attendent les processeurs plus lents. En outre, le langage de programmation doit permettre l'attribution de mémoire dynamique de sorte que la taille exécutable soit proportionnelle à la taille des partitions. La version précédente du code *AERO* emploie le langage de programmation F77 sans allocation de mémoire dynamique de sorte que la taille de l'exécutable est la même pour tous les processeurs et mène à des situations où la RAM de certains processeurs est trop petite pour l'exécution. Dans ce rapport, nous transformons le code F77 *AERO* parallèle en F90 avec l'allocation de mémoire dynamique. La version F90 du code *AERO* est indépendante du maillage et parce que de la mémoire est assignée au temps d'exécution et de la mémoire est seulement assignée pour les options de code réellement utilisées, la taille du F90 exécutable est beaucoup plus petite que la version F77. On présente plusieurs cas tests qui ne peuvent pas être exécutés sur des clusters et des Grilles de calculs avec la version F77 mais peuvent être facilement exécutés avec la version F90. On decrit une application sur un maillage contenant 252K sommets utilisant 8-*nina* et 8-*pf* processeurs sur le MecaGRID avec GLOBUS et des partitions hétérogènes. Le résultat monte un gain de 1,45 par rapport au calcul utilisant une partition homogène. Ceci se compare bien avec le gain théorique de 1,5, et valide l'efficacité du F90 version du code.

Mots-clés : Mécanique des fluides numérique, Grille de calcul, F90

1 Introduction

As CFD software evolves over time, new capabilies are added, for example, moving meshes, two-phase flows, adaption to parallel methods, structure-fluid interaction ... etc. As a consequence the software becomes more and more complex and increasingly difficult for new as well experienced users to manage. In order to increase the efficiency and ease of use, one looks to more advanced progamming languages (F90, C, C++, Java) to achieve these goals.

In this report, the *AERO* software developed over the past 10-15 years to compute fluid-structure interactions is examined [1]. The fluid dynamics part of the *AERO* code used by the author contains 183 subroutines and libraries that exceed 50,000 lines and uses F77 as the programming language.

The object of transforming the F77 version to another programming language is 1) to increase performance[2] and 2) to create a version that is easier for current users and interns [3]to adapt to their research.

With these two stated goals, we need to decide what characteristics the new code should have and choose the language that will provide the desired attributes.

In order to decide what characteristics the new code should have, we first examine the disadvantages of the present F77 version. Then we examine the advantages of choosing F90, C, C++, or Java as the new programming language.

2 Basic assembly loop

Before discussing the parallelization, we describe in this section a basic assembly loop in the software referred to as *AERO* used for one-phase CFD simulations and the *AEDIF* software specific for two-phase flows.

Let $M = \{S, E\}$ be an unstructured mesh as commonly used in finite element and finite volume techniques. $S = \{s_1, \ldots, s_n\}$ denotes the set of the mesh nodes characterized in $3D$ by a three-dimensional point $\underline{x}$. We will denote $(x_i, y_i, z_i), i \in \{1, \ldots, n\}$, the coordinates of the $3D$ point $\underline{x}_i$. Similarly, E denote the set of elements that constitute the mesh. In the *AEDIF* family of codes, we only consider simplicial elements, characterized in $3D$ by 4 nodes.

[1] See [3] [2] [1] [4] for the methods used in the *AERO* code.
[2] Definition to be defined later.
[3] The duration of internships is typically 3-6 months.

Thus the element e will be defined for instance by $e = \{s_i, s_j, s_k, s_l\}$ where i, j, k, l are distinct indices in the set $\{1, \ldots, n\}$. Each element $e \in E$ has four triangular faces $f_j(e), j = 1, 2, 3, 4$. Any face f of any element e is either an **interior** face and therefore belongs also to another element e' :

$\exists\ e \in E$ and $e' \in E$ such that $f = f_j(e) = f_k(e')$ for $j, k \in \{1, 2, 3, 4\}$

or is a **boundary** face :

$\exists!\ e \in E$ and $k \in \{1, 2, 3, 4\}$ such that $f = f_k(e)$.

An element $e \in E$ defined by the four nodes $\{s_i, s_j, s_k, s_l\}$ possesses 6 edges (s_p, s_q) where p, q are two distinct indices in the set $\{i, j, k, l\}$. However any edge a belongs to an arbitrary number of elements and in the same way, the number of neighbors of a node is arbitrary. We will denote by $\mathcal{A}$, the set of all the edges and by $\mathcal{V}(i)$ the set of the neighbors of s_i.

AEDIF uses a vertex centered Finite-Volume method, i.e to any nodes $s_i; i = 1, \ldots, n$ is attached a variable vector v_i of dimension nf, that is an approximation of some physical quantities defined at the coordinates $\underline{x_i}$. For instance, in Computational Fluid Dynamics (CFD) $nf = 5$ and to any node, s_i is attached the vector $\mathbf{Q}_i = (\rho_i, \underline{m}_i, E_i)$ where ρ is the density, $\underline{m}$ the momentum ($3D$ vector) and E the total energy. The vector $\mathbf{Q}_i$ is supposed to be an approximation of the field vector $\mathbf{Q}$ at the position $\underline{x_i}$

$$\mathbf{Q}_i \sim \mathbf{Q}(\underline{x_i})$$

Let $n = 0$ and $\mathbf{Q}^0$ be a given value of the vector $\mathbf{Q}$ defined on all the nodes of the mesh M, Finite Volume methods (FV) will compute an improved value $\mathbf{Q}^{n+1}$ of the field vector on the mesh nodes based on the current value of $\mathbf{Q}_i$ and of its neighbors $\mathbf{Q}_j$ where $j \in \mathcal{V}(i)$. More specifically, a FV scheme is a

recipe implementing the following update of the variables :

$$\begin{aligned} &\text{For} \qquad i = 1, \ldots, n \\ &\qquad \Delta \mathbf{Q}_i \leftarrow \textstyle\sum_{l \in \mathcal{V}(i)} H(\mathbf{Q}_i, \mathbf{Q}_l, G_{il}) \\ &\qquad V_i \times \mathbf{Q}_i^{n+1} \leftarrow V_i \times \mathbf{Q}_i^n + \Delta \mathbf{Q}_i \\ &\text{endfor} \end{aligned} \tag{1}$$

where V_i the volume associated to the node s_i is a function depending only on $\underline{x}_i$ and $\underline{x}_l$ with $l \in \mathcal{V}(i)$, H is a function depending on the three arguments, $\mathbf{Q}_i, \mathbf{Q}_l, G_{il}$ with G_{il} a geometrical factor that depends only on the coordinates $\underline{x}_i$ and $\underline{x}_l$ with $l \in \mathcal{V}(i)$. The important point is that the function H verifies the conservation property :

$$H(b, a, -G) = -H(a, b, G) \tag{2}$$

and thus once, G_{il} are computed, Algorithm 1 can be implemented as follows :

$$\begin{aligned} &\text{For} \qquad a \ \in \mathcal{A} \\ &\qquad \text{Get}\{i, j\} \text{ such that } a = \{i, j\} \\ &\qquad \text{Compute } K = H(\mathbf{Q}_i, \mathbf{Q}_j, G_{ij}) \\ &\qquad \Delta \mathbf{Q}_i \leftarrow \Delta \mathbf{Q}_i + K \\ &\qquad \Delta \mathbf{Q}_j \leftarrow \Delta \mathbf{Q}_j - K \\ &\text{EndFor} \end{aligned} \tag{3}$$

$$\begin{aligned} &\text{For} \qquad i = 1, \ldots, n \\ &\qquad V_i \times \mathbf{Q}_i^{n+1} \leftarrow V_i \times \mathbf{Q}_i^n + \Delta \mathbf{Q}_i \\ &\text{EndFor} \end{aligned}$$

and the function $H(\mathbf{Q}_i, \mathbf{Q}_l, G_{il})$ that is generally costly to compute can be evaluated once by edge. Algorithm 3 is thus less costly by a factor 2 than Algorithm 1.

3 Parallelization in the *AERO* code

As will be seen in section 6, the primary disadvantages of F77 arise due to parallel computation. To better understand why this is the case, we explain how parallel computing is implemented in the *AERO* code.

Before the advent of parallel computers, a single processor performed all the computations. Technology in memory storage and access advanced much more rapidly than the speed of processors, thus it became obvious that for large grids and more sophisticated physical models, a new form of computing was needed. This led to the development of machines that use multiple processors or computing in parallel (now simply referred to as parallel computing). The idea of parallel computing is to divide the total work over multiple processors, thus speeding up the elapsed time between the start and end of the computation.

The *AERO* code implements parallel computing using mesh partitioning (also called domain or block decomposition), that is, dividing a large mesh into smaller partitions (or domains, blocks) so that the work for each partition can be computed by a different processor (or CPU). For example, if 32 processors are used, the mesh is divided into 32 approximately equal partitions (known as homogeneous partitioning) so that each processor does approximately 1/32 of the total work[4]. Using mesh partitioning or domain decomposition is a very popular approach for parallel computing using CFD codes.

Once the mesh has been partitioned, parallelization in the the F77 *AERO* code is implemented using the Message Passing Interface (MPI) system applied over the partitions. The source code with the MPI calls added is compiled with the F77 and MPI libraries creating a executable which we call $aero77.x$. Using MPI, each processor executes a copy of $aero77.x$, using a common data file $flu.data$ that contains data common to all the processors (time scheme, order of accuracy, CFL number, number of time steps, ...) and a $flu.glob$ that

[4]As a side note, since each partition is much smaller than the total mesh, the speed of the processors of parallel machines can be much smaller than machines that use a single very fast processor to perform all the computational work.

contains the size of the full mesh and the maximum size of the partitions. However, each processor reads only the mesh data for its specific partition. The partition data is labeled as flu-0001, flu-00002, ..., flu-0000n. Processor 0 computes using the partition data flu-00001, processor 1 flu-00002, and so forth.

The files flu-000n contain for the partition in question, the number of vertices, tetrahedra, and external faces (triangles); the coordinates of the vertices, the connectivity for the tetrahedra and external faces (triangles), the type of external boundary conditions to be applied on the external faces, the vertices that must exchange data with neighboring partitions, an identifier to indicate whether a boundary vertice is active or not and finally the global indices of the local vertices. APPENDIX A gives additional details of the data contained of the flu-0000n files.

4 Definition of partitioned mesh

We now discuss in greater detail the strategy used for the parallel implementation of algorithm 3. In the parallelization strategy presently used, the domain decomposition is based on the **elements** of the mesh. To be more specific, consider the mesh of figure 1 and assume that we have two processors 1 (black) and 2 (red).

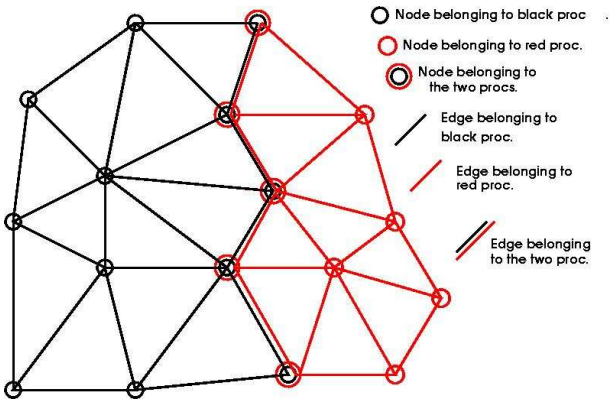

Figure 1: **Data structure for partitioned mesh**

We denote E_1 and E_2 with $E_1 \cup E_2 = E$ and $E_1 \cap E_2 = \emptyset$, the partition of the element set. We then define $S_p, p = 1, 2$ the set of nodes corresponding to the set E_p

$$S_p = \{l \in S; l \in E_p\} \subset S \tag{4}$$

Similarly, we define $\mathcal{A}_p$, the set of edges corresponding to the elements in E_p.

$$\mathcal{A}_p = \{a \in \mathcal{A}; a = (k, l) \text{ with } k \in S_p \text{ and } l \in S_p\} \tag{5}$$

The set $S_p, \mathcal{A}_p, E_p$ will be stored on processor p. This implies (see figure 1) a duplication of some nodes **and** edges of the initial mesh. We note $S_{p,q}$ (resp. $\mathcal{A}_{p,q}$ the set of nodes (resp. edges) that are shared both by processors p and q

$$S_{p,q} = S_p \cap S_q \text{ and } \mathcal{A}_{p,q} = \mathcal{A}_p \cap \mathcal{A}_q \tag{6}$$

In an opposite way, a node l (resp. edge a) such that $l \in S_p - S_{p,q}$ for any q (resp. $a \in \mathcal{A}_p \backslash \mathcal{A}_{p,q}$) will be called an interior point for processor p.

For the parallel implementation of Algorithm 3, we note that $M_p = (S_p, E_p)$, $p = 1, \ldots, P$ constitute P independent meshes, thus if for any $s_i \in S_p$ is defined and stored on processor p a variable $\mathbf{Q}_i^p$, Algorithm 3 can be used independently on each processor to update the variables $\mathbf{Q}_i^p$. This parallel application of Algorithm 3 allows to update all the **interior** points. However, if $i \in S_{p,q}$ the variable $\mathbf{Q}_i^p$ will be updated only from contributions coming from neighbors belonging to the processor p and thus the contribution coming from processor q have to be transmited from this processor and added to the current update. Moreover, we observe that if an edge $a = \{i, l\}$ is in $\mathcal{A}_{p,q}$ the computation of $H(\mathbf{Q}_i, \mathbf{Q}_l, G_{il})$ will be done twice, on the two processors p and q and therefore the sum $\Delta\mathbf{Q}_i^p + \Delta\mathbf{Q}_i^q$ will contain twice this contribution. Thus to avoid this, we define the notion of **active** edge and any edge $a \in \mathcal{A}_{p,q}$ is attributed only to one processor p or q. In summary, the parallel implementation of Algorithm 3 is thus :

$$
\begin{array}{lll}
\text{For} & p = & 1,\ldots P \text{ do in parallel} \\
 & \text{For} & a \in \mathcal{A}_p \\
 & & \text{If } a \text{ is an active edge then} \\
 & & \text{Get}\{i,j\} \text{ such that } a = \{i,j\} \\
 & & \text{Compute } K = H(\mathbf{Q}_i, \mathbf{Q}_j, G_{ij}) \\
 & & \Delta\mathbf{Q}_i^p \leftarrow \Delta\mathbf{Q}_i^p + K \\
 & & \Delta\mathbf{Q}_j^p \leftarrow \Delta\mathbf{Q}_j^p - K \\
 & & \text{EndIf} \\
 & \text{EndFor} & \\
 & \text{For} & i \in S_p \\
 & & V_i \times \mathbf{Q}_i^{n+1} \leftarrow V_i \times \mathbf{Q}_i^n + \Delta\mathbf{Q}_i^p \\
 & \text{EndFor} & \\
\text{EndParallelFor} & &
\end{array}
\tag{7}
$$

$$
\begin{array}{lll}
\text{For} & p = & 1,\ldots P \text{ do in parallel} \\
 & \text{For} & i \in S_{p,q} \\
 & & \text{Get from proc. } q \ \Delta\mathbf{Q}_i^q \\
 & & V_i \times \mathbf{Q}_i^{n+1} \leftarrow V_i \times \mathbf{Q}_i^{n+1} + \Delta\mathbf{Q}_i^q \\
 & \text{EndFor} & \\
\text{EndParallelFor} & &
\end{array}
\tag{8}
$$

5 Why F77 is not suited for GRID computing

For machines like the SGI 3800 at CINES[5] with 768 processors, homogeneous partitioning works fine as all the processors are identical in speed and RAM and therefore finish their work in approximately the same amount of time.

However when the processors have different speeds and RAM, efficiency is lost (example: computing on the INRIA cluster using both "*nina*" and "*pf*" processors - see Table 11). This is also the scenario when computing on

[5]Centre Informatique National de l'Enseignement Supérieur Montpellier, France, http://www.cines.fr

Computational Grids like the MecaGRID[6] that connects the clusters of INRIA Sophia Antipolis, Ecole des Mines de Paris at Sophia Antipolis, and the IUSTI in Marseille. - see Wornom [5]. Table 11 shows that the speed and RAM of the *nina* processors to be twice that of the *pf* processors.

Processor	CPUs	GHz	RAM/Node	LAN speed
nina	32	2	1.00 GB	1.000 Gbps
pf	38	1	0.50 GB	0.100 Gbps

Table 1: Characteristics of the INRIA cluster

Partition 1	Partition 2	Partition 3	Partition 4
ns = 2000 aero.x flu.data flu.glob flu-00001	ns = 2000 aero.x flu.data flu.glob flu-00002	ns = 2000 aero.x flu.data flu.glob flu-00003	ns = 2000 aero.x flu.data flu.glob flu-00004

Table 2: Mesh with 4 equal partitions

Suppose that the 4 equal-partition mesh[7] illustrated in Table 2 is executed with either 4-*nina* processors or with 4-*pf* processors. There are several possible results:

1. If the size of the executable, *aero.x*, is less than 1/2 GB, the calculation will run on either the 4-*nina* processors or 4-*pf* processors with the wall time for the 4-*pf* computation being on the order of twice that for the 4-*nina* computation.
2. If the size of the executable is greater than 1/2 GB but less than 1 GB, the calculation will run on the 4-*nina* processors (1 GB RAM) but not the 4-*pf* processors (1/2 GB RAM).

[6]The MecaGRID project is sponsered by the French Ministry of Research through the ACI-Grid program, http://www.recherche.gouv.fr/recherche/aci/grid.htm.

[7]ns = number of vertices

3. Suppose that the size of the executable is greater than 1/2 GB and combinations of *nina* and *pf* processors are used. The computation will not execute because of the 1/2 GB RAM limit for the *pf* processors.

4. Suppose that the size of the executable is less than 1/2 GB and combinations of *nina* and *pf* processors are used. The computation will execute but will not be efficient as both the *nina* and *pf* processors have the same amount of computational work but the *nina* processor is twice as fast as the *pf* processor, therefore the *nina* processors will wait until the slower *pf* processors have finished. In this case the speedup should be no faster than the case where 4-*pf* processors are used.

In order to fully optimize efficiency, complete load balancing is necessary. Complete load balancing consists of two steps: 1) Give more computational work to the faster processors (larger partitions). For *nina-pf* processors this means:

$$\frac{work_{nina}}{speed_{nina}} = \frac{work_{pf}}{speed_{pf}} \tag{9}$$

or

$$work_{nina} = 2 \cdot work_{pf} \tag{10}$$

and 2) dynamical allocate memory according to the size of the partitions and the code options actually used.

The first part is achieved using a mesh partitioner that partitions the original mesh according to processors speed.

Partition 1	Partition 2	Partition 3
ns = 4000 aero.x flu.data flu.glob flu-00001	ns = 2000 aero.x flu.data flu.glob flu-00002	ns = 2000 aero.x flu.data flu.glob flu-00003

Table 3: Mesh with 3 unequal-partitions

Therefore *nina* processors would get a domain twice the size of the *pf* processors similar to the mesh illustrated in Table 3 with three unequal partitions designed for using 1-*nina* processor and 2-*pf* processors. Step 1 speeds up the computation for the F77 code under the condition that the size of the F77 executable, *aero.x*, is less than the 1/2 GB RAM limit for the *pf* processors. As we shall see, is very important that complete load balancing be used to obtain the maximum efficient.

An alternative novel load balancing approach for the F77 code was suggested by Alain Dervieux and applied by Wornom [5]. The idea is that rather than partition the mesh according to the processor speed, create homogeneous mesh partitions (equal sizes) and give more partitions to the faster processors at execution. This avoids the necessity to run the mesh partitioner each time before executing the *AERO* code [8]. Therefore the *nina* processors would get two partitions and the *pf* processors one partition. Another advantage of this approach is that for small to medium meshes, homogeneous partitions can be configured to fit the minimum RAM available (1/2 GB for *pf* processors).

6 Choice of new programming language

The primary characteristic sort in changing from F77 to a different programming language is the ability to load balance according to processor speed, ie., to use meshes similar to the mesh shown in Table 3. Complete load balancing cannot be achieved using F77 since parameters statements are used and the parameters are fixed not by the size of the local partition but by the size of the largest partition. This rules out complete load balancing. Partial load balancing is possible if the F77 executable will run on the processors with the smallest RAM (*pf* with 1/2 GB for the INRIA cluster).

In order to have complete load balancing, the program language must have dynamic memory allocation (memory is allocated by local data rather than global data). F90, C, C++, and Java all have this property. If we choose either C, C++, or Java as the new language, it is necessary to change 100 percent of the F77 coding as none of these languages is compatible with Fortran. This is very time consuming and the transition time to rewrite the F77 coding in

[8] Recall that the user does not know in advance the mixture for *nina* and *pf* processors that he/she will receive. Therefore the mesh partitioner must be executed on the mixture of *nina* and *pf* processors before running the *AERO* code.

the new language and debug the new code will be long. In addition, Fortran users and developers will be at a great disadvantage with the C, C++, or Java code. For the following reasons, F90 is chosen as the new language:

1. F77 is compatible with the F90 compiler. This means that 1) we can add F90 features to the existing F77 code and 2) that it is not necessary to convert 100 percent of the F77 code to the F90 standard. This saves considerable time both in not having to rewrite the F77 code in the F90 standard as well as the additional debugging time that would be necessary.

2. The F77 code can be easily modified to include F90 dynamic memory allocation feature. In fact, most of the necessary changes were completed over two-day period and an additional week to complete the tests and verification[9].

3. Most of the F90 modifications are transparent to the users, thus the use of the F90 version results in no perturbation to old users.

4. The F77 Makefile depends both on the mesh and the number of processors used and is very complex for new and even experienced users. With the dynamic memory allocation, the F90 Makefile depends only on the source and therefore very easy for new and old users to use.

7 Adding dynamic memory allocation

In this section we show the changes necessary to add the F90 dynamic memory allocation to the F77 code. Note that a subroutine using 100 percent of the F90 standard is written like file.f90. A full F90 standard is not necessary and we use the form file.f form but instead of compiling with F77, we compile with F90.

The F77 code uses parameter statements to tell the F77 compiler the size of the arrays needed at execution time. The parameter statements are defined in one of 12 parameter files (.h files). In order to have dynamic memory allocation,

[9]See APPENDIX B for details

the parameter statements containing arrays and parameters are converted to F90 modules[10] with allocatable arrays[11].

Shown in Table 4 are a few lines from the F77 Param3D.h parameter statement (611 lines total). Table 5 shows the corresponding F90 Module "Module_Param3D.f." As can be seen, the changes are quite simple. Similarly, shown in Table 6 are a few lines from the F77 Paral3D.h parameter statement (161 lines total). Table 6 shows the corresponding F90 Module "Module_Paral3D.f", again the changes are quite simple. The important difference between F77 and F90 is that F77 uses parameters and anytime a parameter is changed the F77 source must be recompiled whereas F90 parameters are set at run time and recompiling the F90 source is only necessary if the source is modified.

Table 8 shows how the parameter statements are included in the source code. The following observations for F77 are noted:

1. The size of the executable is determined at compile time and depends on the parameter statements (.h files). The parameter statements (.h files) are set by the mesh and the number of partitions.

2. The parameter statements are not only mesh dependent but also dependend on the number of processors used. Thus for each problem (mesh), the user must 1) compute the new parameters and 2) change the .h files to correspond to the problem of concern. If the number of processors used changes, the maxthd parameter (number of processors) in the Paral3D.h file must be reset to the correct value. As a consequence, the F77 Makefile(s) are not always easy to use.

3. The *AERO* code uses the MPI system to execute in parallel, using F77, the size of the executable is determined by the size of the largest domain (or partition) and therefore load balancing via different size partitions is not possible.

4. All arrays are allocated memory at compile time whether or not they are use at run time. This results in a very large executable that reduces the size of the mesh that can be executed.

[10] Modules are compile like any other .f files in the code.

[11] Shown in left column of Table 12 in APPENDIX C are the F77 parameter statements with the corresponding F90 Modules in the right column. Six parameter statements were left unchanged as they were independent of the mesh and number of processors being used.

```
c   F77   start              Param3D.h
...
c nsmaxg = Maximum number of vertices in unpartitioned mesh
c nsmax  = Maximum number of local vertices
c ntmax  = Maximum number of local tetrahedra

      INTEGER nsmaxg, nsmax, ntmax

      PARAMETER (nsmaxg = 16420 )
      PARAMETER (nsmax  = 2241 )
      PARAMETER (ntmax  = 10470 )

c     nu      = Tetraedra connectivity table
      INTEGER nu(4,ntmax)
      COMMON /cm01/ nu

c     ua          = number of local variables
c     Waverage = Turbulence array
c     xw          = Mesh vertices velocities

      REAL ua(7,nsmax), Waverage(5,nsmaxg)
      COMMON /cm02/ ua, Waverage, xw
...
c   F77   end              Param3D.h
```

Table 4: F77 Param3D.h parameter.

```
c   F90
      MODULE Param3D
...
c nsmaxg = Maximum number of vertices in unpartitioned mesh
c nsmax  = Maximum number of local vertices
c ntmax  = Maximum number of local tetrahedra

      INTEGER nsmaxg, nsmax, ntmax

c     nu      = Tetraedra connectivity table
      INTEGER,allocatable,dimension (:,:) :: nu

c     ua          = number of local variables
c     Waverage = Turbulence array
c     xw          = Mesh vertices velocities

      REAL,allocatable,dimension (:,:) :: ua, Waverage, xw
...
      END MODULE Param3D
c   F90
```

Table 5: F90 MODULE Param3D.

```
c   F77   start             Paral3D.h
...
c Maximum number of processors for the parallel execution
      INTEGER maxthd
      PARAMETER (maxthd = 16 )
c
c     Direct and inverse mapping of the submeshes on the processing nodes
      INTEGER imapd(maxthd), ivmapd(0:maxthd-1)
      COMMON/com01/imapd, ivmapd
c
c Maximum number of interface edges in common with a neighboring submesh
      INTEGER lsgmax
      PARAMETER (lsgmax = 12255 )
c
      INTEGER insi2t(maxthd), isint(lsgmax,maxthd)
      COMMON /com02/ insint, isint
...
c   F77   end               Paral3D.h
```

Table 6: F77 Paral3D.h parameter statement.

```
c   F90
      MODULE Paral3D
...
c Parameter and common definitions for the parallel implementation

c Maximum number of processors for the parallel execution
      INTEGER maxthd

c     Direct and inverse mapping of the submeshes on the processing nodes
      INTEGER,allocatable,dimension (:)   :: imapd
      INTEGER,allocatable,dimension (:,:) :: ivmapd

c Maximum number of interface edges in common with a neighboring submesh
      INTEGER lsgmax
      INTEGER,allocatable,dimension (:)   :: insint
      INTEGER,allocatable,dimension (:,:) :: isint
...
      END MODULE Paral3D
```

Table 7: F90 MODULE Paral3D.

```
c F77
      SUBROUTINE SUBMSH

      INCLUDE 'Param3D.h'
      INCLUDE 'Paral3D.h'
...
c       ns   = number of local vertices
c       nt   = number of local tethedra
c       nfac = number of local exterior boundary faces
...
       READ(myunit, *) ns, nt, nfac
...
      RETURN
      END
```

Table 8: F77 subroutine SubMsh.f

8 Characteristics of the F90 code

We show parts of the main program Para3D.f and the subroutine SubMsh.f to point out the main differences the the original F77 code version and the F90 compiled version.

Table 9 shows lines from the PROGRAM Para3D.f. After the "CALL GETNDS(maxthd)" statement, all the arrays needing the number of processors (maxthd) are allocated. Recall that with F77 maxthd is given a value in the Paral3D.h parameter statement that must be changed each time the number of processors used changes and the complete source must be recompiled. Using F90 no recompilation is required as parameters are read at run time.

Shown in Table 10 are lines from the F90 subroutine SubMsh.f. Instead of INCLUDE, USE statements are used. Once the partition or local (or partition) values of ns, nt, nfac[12] have been read from the partition files, flu-00001, flu-00002, ..., memory is allocated for that partition. Note that memory is only allocated for arrays that are used. Data options for the *AERO* code are specified in the flu.data file. Two of these options are *ivis* that specifies the type of flow (Euler, turbulent, large eddy simulation (LES) and *idefor* this specifies whether the mesh is fixed or mobile: If *ivis* = 2 (turbulent flow),

[12]ns = the number of mesh vertices, nt = the number of tetrahedras, and nfac, the number of external triangular faces.

```
c F90
c     PROGRAM PARA3D
...
      CALL GETNDS(maxthd)
...
      allocate( imapd(maxthd), ivmapd(0:maxthd-1) )
      allocate( insint(maxthd), isint(lsgmax,maxthd) )
...
      STOP 999
      END
```

Table 9: F90 Program Para3D.f

memory for the the variables ρk and $\rho\epsilon$ is allocated, otherwise no memory is allocated. Likewise if $idefor = 0$ (fixed mesh), memory space for arrays used for moving meshes is not allocated (vertex velocities, face velocities ...). Allocating memory only for options actually used results in a very efficient code as concerns memory allocation and permits computations with much larger meshes particularly when mesh load balancing is used. For temporary arrays, memory is allocated and deallocated when the arrays are no longer needed.

9 Temporary memory

In the F77 code all memory is allocated at compile time. For example, there are 12 subroutines where arrays are allocated memory that is temporary memory but F77 allocate memory as if were permanent memory. An example is

F77
REAL(sbuff(nbvar*nsmax+1))
REAL(rbuff(nbvar*nsmax+1))

in F90, we
allocate(sbuff(nbvar*nsmax+1))
allocate(rbuff(nbvar*nsmax+1))

and
deallocate(sbuff)

```
c F90
      SUBROUTINE SUBMSH

      USE Param3D
      USE Paral3D
...
c       ns   = number of local vertices
c       nt   = number of local tethedra
c       nfac = number of local exterior boundary faces
...
c ivis = 0 Euler
c ivis = 2 Turbulent (K-Epsilon)
c
c nvar = number of variables (5 = default)
      nvar = 5
      if ( ivis .eq. 2 ) nvar = nvar + 2

      READ(myunit, *) ns, nt, nfac

      allocate( ua(nvar,ns) )

      if ( ivis .eq. 2 ) then
         allocate( Waverage(5,nsmaxg)
         ...
      endif

c fixed mesh: idefor = 0
c mobile mesh: idefor .ne. 0
c Mesh vertices velocities
      if ( idefor .ne. 0 ) then
         allocate(xw(6,ns))
          ...
      endif
...
      RETURN
      END
```

Table 10: F90 subroutine SubMsh

deallocate(rbuff)
when we exit the subroutine.

10 Improvements related to F90

INRIA cluster: The problem that motivated the F90 work was a LES (Large Eddy Simulation) test case involving 400K mesh points that could not be run on the INRIA cluster using 32-partitions. The INRIA cluster is really composed of two clusters, one called *nina* and the other *pf*[13]. Table 11 shows that the speed and RAM of the *nina* processors to be twice that of the *pf* processors. Note that *nina* and *pf* both have 32 processors. If ≤ 32 processors are needed, the user can request all *nina* processors or all *pf* processors or have the cluster assign a mixture of *nina* and *pf* processors based on availability. If the user needs more than 32 processors, the user must let the cluster scheduler assign a mixture of *nina* and *pf* processors. The 400K mesh runs requested 32-*nina* processors. As there are only 32-*nina* processors, it is extremely rare that a user will get all 32[14]. Thus the user attempted runs with a mixture of *nina-pf* processors (total of 32).

Processor	CPUs	GHz	RAM/Node	LAN speed
nina	32	2	1.00 GB	1.000 Gbps
pf	32	1	0.50 GB	0.100 Gbps

Table 11: Characteristics of the INRIA cluster

This was due to the size of the F77 executable that was .9 GB, ok for the *nina* processors (1 GB RAM) but too large for the *pf* processors (1/2 GB RAM). In order to run the F77 version, the user accepted to use the explicit algorithm[15].

Using the F90 version, both the explicit and implicit executed successfully using *nina* and *pf* processors for the 400K mesh. Also, based on our tests, we estimate that using the F90 version, a mesh of 1.2 million points could be run using only 32-*pf* processors!

MecaGRID: In order to verify the DMA feature of the F90 code, a test case was run on the MecaGRID using heterogeneous mesh partitioning. The test

[13] See http://www-sop.inria.fr/parallel/

[14] These runs were requested during a period in which most of the 32-*nina* processors were not available.

[15] The 400K grid has refinement near mesh near boundaries; therefore the implicit algorithm was preferred to avoid the long computational times needed with the explicit algorithm. Using the explicit algorithm permitted us to remove eight implicit subroutines from the Makefile and their call statements from the source code to get the size of the executable $\leq$ 1/2 GB.

case studied is a plane shockwave propagating in the z-direction of a rectangular tube. The mesh contains 252K vertices (51x51x97 in the x, y, z directions). Two partitionings were created: 1) 16 homogeneous partitions and 2) 16 heterogeneous partitions (8-*nina* 8-*pf*). The mesh partitioner used in this study was the FastMP software developed by Wornom and Dervieux [6]. The FastMP software is written in F90 and executes in parallel using MPI and is quite fast. The heterogeneous partitioning for the 252K mesh required 3.8 seconds[16].

We executed the F90 code on the MecaGRID using the GLOBUS software[17] with a total of 16 processors (8-*nina* and 8-*pf*). 150 time steps were computed. MecaGRID users can select the number of processors on each cluster that of the MecaGRID. Both the *nina* and *pf* clusters are members of the MecaGRID. For the homogeneous partitioning, all the partitions were of equal size. For the heterogeneous mesh, the 8-*nina* partitions contained 23409 vertices, the 8-*pf* partitions contained 13005 vertices (1/2 as many). The maximum time for all processors was 701 seconds for the homogeneous partitioning and 484 seconds for the heterogeneous partitioning, thus a speedup of 1.45 using the heterogeneous partitioning (the theoretical speedup for this case is 1.5).

11 Conclusions

F90 offers several features not available in F77 that increase the capabilities and ease of use of the *AERO* code. These include dynamic memory allocation. The F90 version of the *AERO* code is mesh independent and because memory is allocated at runtime, only arrays and options that are used are allocated memory. Therefore the size of the executable is much smaller than the F77 version that allocates memory for all options, whether they are used are not, at run time. As a consequence many test cases (large meshes) that cannot be run with the F77 version can be easily run with the F90 version. This is particularily useful for Computational GRIDS. Numerical results for a mesh containing 252K vertices using 8-*nina* and 8-*pf* processors running on the MecaGRID using GLOBUS using GLOBUS and heterogeneous partitions resulted in a speedup of 1.45 relative to the same run using the homogeneous partitioning which compares well with the theoretical speedup of 1.5. This validates the efficiency of the F90 version of the *AERO* code.

[16]Time to read the mesh to be partitioned, partition the mesh and write the partition files
[17]http://www.globus.org

12 Acknowledgements

The author would like to thank Hervé Guillard and Alain Dervieux for their support of this work. Special thanks to Hervé Guillard for his contributions to Sections 2 and section 4. The authors would like to acknowledge the support of Centre Informatique National de l'Enseignement Supérieur (CINES[18]), Montpellier, FRANCE.

[18] http://www.cines.fr

References

[1] B. Nkonga and H. Guillard. Godunov type method on non-structured meshes for three dimensional moving boundary problems. *Comput. Methods Appl. Mech. Eng*, 113:183–204, 1994.

[2] A. Dervieux. Steady Euler simulations using unstructured meshes. Lecture series 1985-04[19], Von Karman Institute for Fluid Dynamics, 1985.

[3] C. Farhat. High performance simulation of coupled nonlinear transient aeroelastic problems. Special course on parallel computing in CFD. R-807, NATO AGARD Report, October 1995.

[4] R. Martin and H. Guillard. Second-order defect-correction scheme for unsteady problems. *Computer & Fluids.*, 25(1):9–27, 1996.

[5] S. Wornom. Parallel computations of one-phase and two-phase flows using the MecaGRID. Technical Report RT-297[20], INRIA - Sophia Antipolis, August 2004.

[6] S. Wornom and A. Dervieux. FastMP, a Fast Mesh Partitioner designed for GRID Computing. Technical report, INRIA - Sophia Antipolis, June 2005.

[19] Published in Partial Differential Equations of hyperbolique type and Applications, Geymonat Ed., World Scientific (1987)

[20] Available at http://www-sop.inria.fr/smash/personnel/Stephen.Wornom/RT-0297.pdf

APPENDIX A

Description of the *flu-0000n* files.

In the flu-0000n we find successively:

ipd = subdomain number (involved by this file flu-xxxxx)
ns, nt, nfac = local number of nodes, tetrahedra and boundary facets
(local means associated with the considered subdomain)
nghd = number of neighboring subdomains

For i=1,nghd we read:

ishd(i) = identification (number) of the ith neighboring subdomain
insghd(ishd(i)) = number of nodes located on the common interface between
the considered subdomain and its ith neighboring subdomain
For ii=1,insghd(ishd(i)) we read:
isghd(ii,ishd(i)) = local number of these common nodes
EndFor ii

EndFor i

For is=1,ns we read:
coor(1,is), coor(2,is), coor(3,is) = x-,y- and z-coordinate of node is
EndFor is

For jt=1,nt we read:
nu(1,jt), nu(2,jt), nu(3,jt), nu(4,jt) = local number of the 4 vertices
of tetrahedron jt
EndFor jt

For ifac=1,nfac we read
logfac(ifac) = boundary identification for each boundary facet ifac
nsfac(1,ifac), nsfac(2,ifac), nsfac(3,ifac) = local number of the 3 vertices
of boundary facet ifac

EndFor ifac

For is=1,ns
irefd(is) = 1 if node "is" is an internal node for the considered subdomain,
0 else
(by internal node, we mean a node which does not belong to the common interfaces shared with neighboring subdomains)
EndFor is

For is=1,ns
igrefd(is) = global number (i.e. number in the global mesh) of node is
EndFor is

REMARK = all the previous variables are integer except coor(,) which is real.

APPENDIX B

F77 parameter statements F90 Modules

F77 parameter statements	F90 Modules
Paral3D.h	Module_Paral3D.f
Param3D.h	Module_Param3D.f
Avepre.h	Module_Avepre.f
Movlog.h	Module_Movlog.f
Visc.h	Module_Visc.f
ToGmresASR.h	Module_ToGmresASR.f
Algtyp.h	Algtyp.h
Mestyp.h	Modes.h
Modes.h	Modes.h
Mulcub.h	Mulcub.h
Rcvs.h	Rcvs.h
Snds.h	Snds.h
	Module_Isegii.f
	Module_Neighboring_submeshes.f
	Module_Stuffcomm.f
	Module_Stuffcomm_mu.f

Table 12: Parameter statement and their Module names

APPENDIX C

Debugging

With the following exceptions, the F77 code was able to be compiled with F90.

1. F77 accepts integers in logical statements[21], F90 does not.
2. F77 accepts "call flush(unit)", SGI F90 requires two arguments "call flush(unit,istat)."
3. F77 accepts multiple quotes in comment statements, F90 does not. WRITE(6, *) 'Roe's approximate Riemann solver'
   this was changed to
   WRITE(6, *) "Roe's approximate Riemann solver"

[21] Example, if(barflag) then, where barflag is defined as an integer. In F90, barflag must be declared as a logical

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803